Chapter 5

The Instruction Set

This chapter describes instruction notation and discusses assembler instructions for the main processor. Chapter 6 describes coprocessor notation and instructions.

Instruction Classes

The assembler has these classes of instructions for the main processor:

- **Load and Store Instructions.** These instructions load immediate values and move data between memory and general registers.
- **Computational Instructions.** These instructions do arithmetic and logical operations for values in registers**.**
- **Jump and Branch Instructions.** These instructions change program control flow.

In addition, there are two other classes of instruction:

- **Coprocessor Interface.** These instructions provide standard interfaces to the coprocessors.
- **Special Instructions.** These instructions do miscellaneous tasks**.**

Reorganization Constraints and Rules

To maximize performance, the goal of RISC designs is to achieve an execution rate of one machine cycle per instruction. When writing assembly language instructions, you must be aware of the rules to achieve this goal. This information is given in the *MIPS R4000 Microprocessor User's Manual* (published by Prentice Hall) or the *MIPS R8000 Microprocessor User's Manual*, depending on which architecture you are using.

Instruction Notation

The tables in this chapter list the assembler format for each load, store, computational, jump, branch, coprocessor, and special instruction. The format consists of an op-code and a list of operand formats. The tables list groups of closely related instructions; for those instructions, you can use any op-code with any specified operand.

Operands can take any of these formats:

- Memory references. For example, a *relocatable symbol* + / – an *expression*(*register*).
- Expressions (for immediate values).
- Two or three operands. For example, ADD $3,$4 is the same as ADD $3,$3,$4.

The operands in the table in this chapter have the following meanings

Operand	Description
address	Symbolic expression (see Chapter2)
breakcode	Value that determines the break
destination	Destination register
destination / src1	Destination register is also source register 1
dest-copr	Destination coprocessor register
dest-gpr	Destination general register
expression	Absolute value
immediate	Expression with an immediate value
label	Symbolic label
operation	Coprocessor-specific operation
return	Register containing the return address
source	Source register
src1, src2	Source registers
src-copr	Coprocessor register from which values are assigned
src-gpr	General register from which values are assigned
target	Register containing the target
z	Coprocessor number in the range 0..2

Instruction Set

The tables in this section summarize the assembly language instruction set. Most of the assembly language instructions have direct machine equivalents.

Load and Store Instructions

Load and store are immediate type intructions that move data between memory and the general registers. Table 5-1 summarizes the load and store instruction format, and Table 5-2 and Table 5-3 provide more detailed descriptions for each load instruction. Table 5-4 and Table 5-5 provide details of each store instruction.

Table 5-1 Load and Store Format Summary

Description	Op-code	Operands
Load Address	LA	*destination, address*
Load Doubleword Address	DLA	
Load Byte	LB	
Load Byte Unsigned	LBU	
Load Halfword	LH	
Load Halfword Unsigned	LHU	
Load Linked*	LL	
Load Word	LW	
Load Word Left	LWL	
Load Word Right	LWR	
Load Doubleword	LD	
Unaligned Load Halfword	ULH	
Unaligned Load Halfword Unsigned	ULHU	
Unaligned Load Word	ULW	
Load Immediate	LI	*destination, expression*

Table 5-1 (continued) Load and Store Format Summary

Description	Op-code	Operands
Load Doublewod Immediate	DLI	
Store Double Right	SDR	
Unaligned Store Doubleword	USD	
Load Upper Immediate	LUI	
Store Byte	SB	*source, address*
Store Conditional *	SC	
Store Double	SD	
Store Halfword	SH	
Store Word Left	SWL	
Store Word Right	SWR	
Store Word	SW	
Unaligned Store Halfword	USH	
Unaligned Store Word	USW	
Load Doubleword	LD	*destination, address*
Load Linked Doubleword	LLD	
Load Word Unsigned	LWU	
Load Doubleword Left	LDL	
Load Doubleword Right	LDR	
Unaligned Load Double	ULD	
Store Doubleword	SD	*source, address*
Store Conditional Doubleword	SCD	
Store Double Left	SDL	

* Not valid in MIPS1 architectures

Load Instruction Descriptions

For all load instructions, the effective address is the 32-bit twos-complement sum of the contents of the index-register and the (sign-extended) 16-bit offset. Instructions that have symbolic labels imply an index register, which the assembler determines. The assembler supports additional load instructions, which can produce multiple machine instructions.

Note: Load instructions can generate many code sequences for which the link editor must fix the address by resolving external data items.

Table 5-2 Load Instruction Descriptions

Instruction Name	Description
Load Address (LA)	Loads the destination register with the effective 32-bit address of the specified data item.
Load Doubleword Address (DLA)	Loads the destination register with the effective 64-bit address of the specified data item (MIPS4 only).
Load Byte (LB)	Loads the least-significant byte of the destination register with the contents of the byte that is at the memory location specified by the effective address. The system treats the loaded byte as a signed value: bit seven is extended to fill the three most-significant bytes.
Load Byte Unsigned (LBU)	Loads the least-significant byte of the destination register with the contents of the byte that is at the memory location specified by the effective address. Because the system treats the loaded byte as an unsigned value, it fills the three most-significant bytes of the destination register with zeros.
Load Halfword (LH)	Loads the two least-significant bytes of the destination register with the contents of the halfword that is at the memory location specified by the effective address. The system treats the loaded halfword as a signed value. If the effective address is not even, the system signals an address error exception.

Table 5-2 (continued) Load Instruction Descriptions

Instruction Name	Description
Load Halfword Unsigned (LHU)	Loads the least-significant bits of the destination register with the contents of the halfword that is at the memory location specified by the effective address. Because the system treats the loaded halfword as an unsigned value, it fills the two most-significant bytes of the destination register with zeros. If the effective address is not even, the system signals an address error exception.
Load Linked (LL) *	Loads the destination register with the contents of the word that is at the memory location. This instruction performs an SYNC operation implicitly; all loads and stores to shared memory fetched prior to the LL must access memory before the LL, and loads and stores to shared memory fetched subsequent to the LL must access memory after the LL. Load Linked and Store Conditional can be use to update memory locations atomically. The system signals an address exception when the effective address is not divisible by four. *This instruction is not valid in the MIPS1 architectures.
Load Word (LW)	Loads the destination register with the contents of the word that is at the memory location. The system replaces all bytes of the register with the contents of the loaded word. The system signals an address error exception when the effective address is not divisible by four.
Load Word Left (LWL)	Loads the sign; that is, Load Word Left loads the destination register with the most-significant bytes of the word specified by the effective address. The effective address must specify the byte containing the sign. In a big-endian system, the effective address specifies the lowest numbered byte; in a little-endian system, the effective address specifies the highest numbered byte. Only the bytes which share the same aligned word in memory are merged into the destination register.

Table 5-2 (continued) Load Instruction Descriptions

Instruction Name	Description
Load Word Right (LWR)	Loads the lowest precision bytes; that is, Load Word Right loads the destination register with the least-significant bytes of the word specified by the effective address. The effective address must specify the byte containing the least-significant bits. In a big-endian configuration, the effective address specifies the highest numbered byte; in a little-endian configuration, the effective address specifies the lowest numbered byte. Only the bytes which share the same aligned word in memory are merged into the destination register.
Load Doubleword (LD)	LD is a machine instruction in the MIPS3 architecture. For the **-mips1** [default] and **-mips2** option: Loads the register pair (*destination* and *destination +1*) with the two successive words specified by the address. The destination register must be the even register of the pair. When the address is not on a word boundary, the system signals an address error exception. **Note:** This is retained for use with the **-mips1** and **-mips2** options to provide backward compatibility only.
Unaligned Load Halfword (ULH)	Loads a halfword into the destination register from the specified address and extends the sign of the halfword. Unaligned Load Halfword loads a halfword regardless of the halfword's alignment in memory.
Unaligned Load Halfword Unsigned (ULHU)	Loads a halfword into the destination register from the specified address and zero extends the halfword. Unaligned Load Halfword Unsigned loads a halfword regardless of the halfword's alignment in memory.
Unaligned Load Word (ULW)	Loads a word into the destination register from the specified address. Unaligned Load Word loads a word regardless of the word's alignment in memory.
Load Immediate (LI)	Loads the destination register with the 32-bit value of an expression that can be computed at assembly time. **Note:** Load Immediate can generate any efficient code sequence to put a desired value in the register.

Table 5-2 (continued) Load Instruction Descriptions

Instruction Name	Description
Load Doubleword Immediate (DLI)	Loads the destination register with the 64-bit value of an expression that can be computed at assembly time. **Note:** Load Immediate can generate any efficient code sequence to put a desired value in the register (MIPS4 only).
Load Upper Immediate (LUI)	Loads the most-significant half of a register with the expression's value. The system fills the least-significant half of the register with zeros. The expression's value must be in the range –32768...65535.

Table 5-3 Load Instruction Descriptions for MIPS3/4 Architecture Only

Instruction Name	Description
Load Doubleword (LD)	Loads the destination register with the contents of the doubleword that is at the memory location. The system replaces all bytes of the register with the contents of the loaded doubleword. The system signals an address error exception when the effective address is not divisible by eight.
Load Linked Doubleword (LLD)	Loads the destination register with the contents of the doubleword that is currently in the memory location. This instruction performs a SYNC operation implicitly. Load Linked Doubleword and Store Conditional Doubleword can be used to update memory locations atomically.
Load Word Unsigned (LWU)	Loads the least-significant bits of the destination register with the contents of the word (32 bits) that is at the memory location specified by the effective address. Because the system treats the loaded word as an unsigned value, it fills the four most-significant bytes of the destination register with zeros. If the effective address is not divisible by four, the system signals an address error exception.

Table 5-3 (continued) Load Instruction Descriptions for MIPS3/4 Architecture Only

Instruction Name	Description
Load Doubleword Left (LDL)	Loads the destination register with the most-significant bytes of the doubleword specified by the effective address. The effective address must specify the byte containing the sign. In a big-endian configuration, the effective address specifies the lowest numbered byte; in a little-endian machine, the effective address specifies the highest numbered byte. Only the bytes which share the same aligned doubleword in memory are merged into the destination register.
Load Doubleword Right (LDR)	Loads the destination register with the least-significant bytes of the doubleword specified by the effective address. The effective address must specify the byte containing the least-significant bits. In a bid-endian machine, the effective address specifies the highest numbered byte. In a little-endian machine, the effective address specifies the lowest numbered byte. Only the bytes which share the same aligned doubleword in memory are merged into the destination register.
Unaligned Load Doubleword (ULD)	Loads a doubleword into the destination register from the specified address. ULD loads a doubleword regardless of the doubleword's alignment in memory.

Store Instruction Descriptions

For all machine store instructions, the effective address is the 32-bit twos-complement sum of the contents of the index-register and the (sign-extended) 16-bit offset. The assembler supports additional store instructions, which can produce multiple machine

instructions. Instructions that have symbolic labels imply an index-register, which the assembler determines.

Table 5-4 Store Instruction Descriptions

Instruction Name	Description
Store Byte (SB)	Stores the contents of the source register's least-significant byte in the byte specified by the effective address.
Store Conditional* (SC)	Stores the contents of a word from the source register into the memory location specified by the effective address. This instruction implicitly performs a SYNC operation; all loads and stores to shared memory fetched prior to the sc must access memory before the sc, and loads and stores to shared memory fetched subsequent to the sc must access memory after the sc. If any other processor or device has modified the physical address since the time of the previous Load Linked instruction, or if an RFE or ERET instruction occurs between the Load Linked and this store instruction, the store fails. The success or failure of the store operation (as defined above) is indicated by the contents of the source register after execution of the instruction. A successful store sets it to 1; and a failed store sets it to 0. The machine signals an address exception when the effective address is not divisible by four. *This instruction is not valid in the MIPS1 architectures.
Store Doubleword (SD)	SD is a machine instruction in the MIPS3 architecture. For the **-mips1** [default] and **-mips2** options: Stores the contents of the register pair in successive words, which the address specifies. The source register must be the even register of the pair, and the storage address must be word aligned. **Note**: This is retained for use with the **-mips1** and **-mips2** options to provide backward compatibility only.
Store Halfword (SH)	Stores the two least-significant bytes of the source register in the halfword that is at the memory location specified by the effective address. The effective address must be divisible by two; otherwise the machine signals an address error exception.

Table 5-4 (continued) Store Instruction Descriptions

Instruction Name	Description
Store Word Left (SWL)	Stores the most-significant bytes of a word in the memory location specified by the effective address. The contents of the word at the memory location, specified by the effective address, are shifted right so that the leftmost byte of the unaligned word is in the addressed byte position. The stored bytes replace the corresponding bytes of the effective address. The effective address's last two bits determine how many bytes are involved.
Store Word Right (SWR)	Stores the least-significant bytes of a word in the memory location specified by the effective address. The contents of the word at the memory location, specified by the effective address, are shifted left so that the right byte of the unaligned word is in the addressed byte position. The stored bytes replace the corresponding bytes of the effective address. The effective address's last two bits determine how many bytes are involved.
Store Word (SW)	Stores the contents of a word from the source register in the memory location specified by the effective address. The effective address must be divisible by four; otherwise the machine signals an address error exception.
Unaligned Store Halfword (USH)	Stores the contents of the two least-significant bytes of the source register in a halfword that the address specifies. The machine does not require alignment for the storage address.
Unaligned Store Word (USW)	Stores the contents of the source register in a word specified by the address. The machine does not require alignment for the storage address.

Table 5-5 Store Instruction Descriptions for MIPS3/4 Architecture Only

Instruction Name	Description
Store Doubleword (SD)	Stores the contents of a doubleword from the source register in the memory location specified by the effective address. The effective address must be divisible by eight, otherwise the machine signals an address error exception.
Store Conditional Doubleword (SCD)	Stores the contents of a doubleword from the source register into the memory locations specified by the effective address. This instruction implicitly performs a SYNC operation. If any other processor or device has modified the physical address since the time of the previous Load Linked instruction, or if an ERET instruction occurs between the Load Linked instruction and this store instruction, the store fails and is inhibited from taking place. The success or failure of the store operation (as defined above) is indicated by the contents of the source register after execution of this instruction. A successful store sets it to 1; and a failed store sets it to 0. The machine signals an address exception when the effective address is not divisible by eight.
Store Doubleword Left (SDL)	Stores the most-significant bytes of a doubleword in the memory location specified by the effective address. It alters only the doubleword in memory which contains the byte indicated by the effective address.
Store Doubleword Right (SDR)	Stores the least-significant bytes of a doubleword in the memory location specified by the effective address. It alters only the doubleword in memory which contains the byte indicated by the effective address.
Unaligned Store Doubleword (USD)	Stores the contents of the source register in a doubleword specified by the address. The machine does not require alignment for the storage address.

Computational Instructions

The machine has general-purpose and coprocessor-specific computational instructions (for example, the floating-point coprocessor). This part of the book describes general-purpose computational instructions.

Computational Instructions

Computational instructions perform the following operations on register values;

- arithmetic
- logical
- shift
- multiply
- divide

Table 5-6 summarizes the computational format summaries, and Table 5-7 and Table 5-8 describe these instructions in more detail.

Table 5-6 Computational Format Summaries

Description	Op-code	Operand
Add with Overflow	ADD	*destination, src1, src2*
Add without Overflow	ADDU	*destination, src1, src2*
AND	AND	*destination, src1, immediate*
Divide Signed	DIV	*destination / src1, immediate*
Divide Unsigned	DIVU	
Exclusive-OR	XOR	
Multiply	MUL	
Multiply with Overflow	MULO	
Multiply with Overflow Unsigned	MULOU	
NOT OR	NOR	
OR	OR	
Set Equal	SEQ	
Set Greater Than	SGT	
Set Greater / Equal	SGE	

Table 5-6 (continued) Computational Format Summaries

Description	Op-code	Operand
Set Greater / Equal Unsigned	SGEU	
Set Greater Unsigned	SGTU	
Set Less Than	SLT	
Set Less / Equal	SLE	
Set Less / Equal Unsigned	SLEU	
Set Less Than Unsigned	SLTU	
Set Not Equal	SNE	
Subtract with Overflow	SUB	
Subtract without Overflow	SUBU	
Remainder Signed	REM	
Remainder Unsigned	REMU	
Rotate Left	ROL	
Rotate Right	ROR	
Shift Right Arithmetic	SRA	
Shift Left Logical	SLL	
Shift Right Logical	SRL	
Absolute Value	ABS	*destination, src1*
Negate with Overflow	NEG	*destination / src1*
Negate without Overflow	NEGU	
NOT	NOT	
Move	MOVE	*destination, src1*
Move Conditional on Not Zero	MOVN	*destination, src1, src2*
Move Conditional on Zero	MOVZ	
Multiply	MULT	*src1,src2*

Table 5-6 (continued) Computational Format Summaries

Description	Op-code	Operand
Multiply Unsigned	MULTU	
Trap if Equal	TEQ	*src1, src2*
Trap if not Equal	TNE	*src1, immediate*
Trap if Less Than	TLT	
Trap if Less than, Unsigned	TLTU	
Trap if Greater Than or Equal	TGE	
Trap if Greater than or Equal, Unsigned	TGEU	
Doubleword Add with Overflow	DADD	*destination,src1, src2* *destination / src1,src2*
Doubleword Add without Overflow	DADDU	*destination, src1, immediate* *destination / src1, immediate*
Doubleword Divide Signed	DDIV	
Doubleword Divide Unsigned	DDIVU	
Doubleword Multiply	DMUL	
Doubleword Multiply with Overflow	DMULO	
Doubleword Multiply with Overflow Unsigned	DMULOU	
Doubleword Subtract with Overflow	DSUB	
Doubleword Subtract without Overflow	DSUBU	

Description	Op-code	Operand
Doubleword Remainder Signed	DREM	
Doubleword Remainder Unsigned	DREMU	

Description	Op-code	Operand
Doubleword Rotate Left	DROL	
Doubleword Rotate Right	DROR	
Doubleword Shift Right Arithmetic	DSRA	
Doubleword Shift Left Logical	DSLL	
Doubleword Shift Right Logical	DSRL	
Doubleword Absolute Value	DABS	*destination*, *src1*
Doubleword Negate with Overflow	DNEG	*destination* / *src1*
Doubleword Negate without Overflow	DNEGU	
Doubleword Multiply	DMULT	*src1*, *src2*
Doubleword Multiply Unsigned	DMULTU	*src1*, *immediate*

Computational Instruction Descriptions

Table 5-7 Computational Instruction Descriptions

Instruction Name	Description
Absolute Value (ABS)	Computes the absolute value of the contents of *src1* and puts the result in the destination register. If the value in *src1* is –2147483648, the machine signals an overflow exception.
Add with Overflow (ADD)	Computes the twos-complement sum of two signed values. This instruction adds the contents of *src1* to the contents of *src2*, or it can add the contents of *src1* to the immediate value. Add (with Overflow) puts the result in the destination register. When the result cannot be extended as a 32-bit number, the machine signals an overflow exception.
Add without Overflow (ADDU)	Computes the twos-complement sum of two 32-bit values. This instruction adds the contents of *src1* to the contents of *src2*, or it can add the contents of *src1* to the immediate value. Add (without Overflow) puts the result in the destination register. Overflow exceptions never occur.

Table 5-7 (continued) Computational Instruction Descriptions

Instruction Name	Description
AND (AND)	Computes the Logical AND of two values. This instruction ANDs (bit-wise) the contents of *src1* with the contents of *src2*, or it can AND the contents of *src1* with the immediate value. The immediate value is not sign extended. AND puts the result in the destination register.
Divide Signed (DIV)	Computes the quotient of two values. Divide (with Overflow) treats *src1* as the dividend. The divisor can be *src2* or the immediate value. The instruction divides the contents of *src1* by the contents of *src2*, or it can divide *src1* by the immediate value. It puts the quotient in the destination register. If the divisor is zero, the machine signals an error and may issue a BREAK instruction. The DIV instruction rounds toward zero. Overflow is signaled when dividing –2147483648 by –1. The machine may issue a BREAK instruction for divide-by-zero or for overflow. **Note:** The special case DIV $0,*src1*,*src2* generates the real machine divide instruction and leaves the result in the HI/LO register. The HI register contains the remainder and the LO register contains the quotient. No checking for divide-by-zero is performed.
Divide Unsigned (DIVU)	Computes the quotient of two unsigned 32-bit values. Divide (unsigned) treats *src1* as the dividend. The divisor can be *src2* or the immediate value. This instruction divides the contents of *src1* by the contents of *src2*, or it can divide the contents of *src1* by the immediate value. Divide (unsigned) puts the quotient in the destination register. If the divisor is zero, the machine signals an exception and may issue a BREAK instruction. See the note for DIV concerning $0 as a destination. Overflow exceptions never occur.
Exclusive-OR (XOR)	Computes the XOR of two values. This instruction XORs (bit-wise) the contents of *src1* with the contents of *src2*, or it can XOR the contents of *src1* with the immediate value. The immediate value is not sign extended. Exclusive-OR puts the result in the destination register.
Move (MOVE)	Moves the contents of *src1* to the destination register.

Table 5-7 (continued) Computational Instruction Descriptions

Instruction Name	Description
Move Conditional on Not Zero (MOVN)	Conditionally moves the contents of *src1* to the destination register after testing that *src2* is not equal to zero (MIPS4 only.)
Move Conditional on Zero (MOVZ)	Conditionally moves the contents of *src1* to the destination register after testing that *src2* is equal to zero (MIPS4 only).
Multiply (MUL)	Computes the product of two values. This instruction puts the 32-bit product of *src1* and *src2*, or the 32-bit product of *src1* and the immediate value, in the destination register. The machine does not report overflow. **Note:** Use MUL when you do not need overflow protection: it's often faster than MULO and MULOU. For multiplication by a constant, the MUL instruction produces faster machine instruction sequences than MULT or MULTU instructions can produce.
Multiply (MULT)	Computes the 64-bit product of two 32-bit signed values. This instruction multiplies the contents of *src1* by the contents of *src2* and puts the result in the HI and LO registers (see Chapter 1). No overflow is possible. **Note:** The MULT instruction is a real machine language instruction.
Multiply Unsigned (MULTU)	Computes the product of two unsigned 32-bit values. It multiplies the contents of *src1* and the contents of *src2* and puts the result in the HI and LO registers (see Chapter 1). No overflow is possible. **Note:** The MULTU instruction is a real machine language instruction.

Table 5-7 (continued) Computational Instruction Descriptions

Instruction Name	Description
Multiply with Overflow (MULO)	Computes the product of two 32-bit signed values. Multiply (with Overflow**)** puts the 32-bit product of *src1* and *src2*, or the 32-bit product of *src1* and the immediate value, in the destination register. When an overflow occurs, the machine signals an overflow exception and may execute a BREAK instruction. **Note:** For multiplication by a constant, MULO produces faster machine instruction sequences than MULT or MULTU can produce; however, if you do not need overflow detection, use the MUL instruction. It's often faster than MULO.
Multiply with Overflow Unsigned (MULOU)	Computes the product of two 32-bit unsigned values. Multiply (with Overflow Unsigned**)** puts the 32-bit product of *src1* and *src2*, or the product of *src1* and the immediate value, in the destination register. This instruction treats the multiplier and multiplicand as 32-bit unsigned values. When an overflow occurs, the machine signals an overflow exception and may issue an BREAK instruction. **Note:** For multiplication by a constant, MULOU produces faster machine instruction sequences than MULT or MULTU can reproduce; however, if you do not need overflow detection, use the MUL instruction. It's often faster than MULOU.
Negate with Overflow (NEG)	Computes the negative of a value. This instruction negates the contents of *src1* and puts the result in the destination register. If the value in *src1* is –2147483648, the machine signals an overflow exception.
Negate without Overflow (NEGU)	Negates the integer contents of *src1* and puts the result in the destination register. The machine does not report overflows.
NOT (NOT)	Computes the Logical NOT of a value. This instruction complements (bit-wise) the contents of *src1* and puts the result in the destination register.
NOT OR (NOR)	Computes the NOT OR of two values. This instruction combines the contents of *src1* with the contents of *src2* (or the immediate value). NOT OR complements the result and puts it in the destination register.

Table 5-7 (continued) Computational Instruction Descriptions

Instruction Name	Description
OR (OR)	Computes the Logical OR of two values. This instruction ORs (bit-wise) the contents of *src1* with the contents of *src2*, or it can OR the contents of *src1* with the immediate value. The immediate value is not sign-extended. OR puts the result in the destination register.
Remainder Signed (REM)	Computes the remainder of the division of two unsigned 32-bit values. The machine defines the remainder REM(*i*,*j*) as *i*–(*j**div(*i*,*j*)) where *j* · 0. Remainder (with Overflow) treats *src1* as the dividend. The divisor can be *src2* or the immediate value. This instruction divides the contents of *src1* by the contents of *src2*, or it can divide the contents of *src1* by the immediate value. It puts the remainder in the destination register. The REM instruction rounds toward zero, rather than toward negative infinity. For example, div(5,–3)=–1, and rem(5,–3)=2. For divide-by-zero, the machine signals an error and may issue a BREAK instruction.
Remainder Unsigned (REMU)	Computes the remainder of the division of two unsigned 32-bit values. The machine defines the remainder REM(*i*,*j*) as *i*–(*j**div(*i*,*j*)) where *j* · 0. Remainder (unsigned) treats *src1* as the dividend. The divisor can be *src2* or the immediate value. This instruction divides the contents of *src1* by the contents of *src2*, or it can divide the contents of *src1* by the immediate value. Remainder (unsigned) puts the remainder in the destination register. For divide-by-zero, the machine signals an error and may issue a BREAK instruction.
Rotate Left (ROL)	Rotates the contents of a register left (toward the sign bit). This instruction inserts in the least-significant bit any bits that were shifted out of the sign bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. Rotate Left puts the result in the destination register. If *src2* (or the immediate value) is greater than 31, *src1* shifts by (*src2* MOD 32).

Table 5-7 (continued) Computational Instruction Descriptions

Instruction Name	Description
Rotate Right (ROR)	Rotates the contents of a register right (toward the least-significant bit). This instruction inserts in the sign bit any bits that were shifted out of the least-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. Rotate Right puts the result in the destination register. If *src2* (or the immediate value) is greater than 32, *src1* shifts by *src2* MOD 32.
Set Equal (SEQ)	Compares two 32-bit values. If the contents of *src1* equal the contents of *src2* (or *src1* equals the immediate value) this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Greater Than (SGT)	Compares two signed 32-bit values. If the contents of *src1* are greater than the contents of *src2* (or *src1* is greater than the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Greater / Equal (SGE)	Compares two signed 32-bit values. If the contents of *src1* are greater than or equal to the contents of *src2* (or *src1* is greater than or equal to the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Greater / Equal Unsigned (SGEU)	Compares two unsigned 32-bit values. If the contents of *src1* are greater than or equal to the contents of *src2* (or *src1* is greater than or equal to the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Greater Than Unsigned (SGTU)	Compares two unsigned 32-bit values. If the contents of *src1* are greater than the contents of *src2* (or *src1* is greater than the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.

Table 5-7 (continued) Computational Instruction Descriptions

Instruction Name	Description
Set Less Than (SLT)	Compares two signed 32-bit values. If the contents of *src1* are less than the contents of *src2* (or *src1* is less than the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Less / Equal (SLE)	Compares two signed 32-bit values. If the contents of *src1* are less than or equal to the contents of *src2* (or *src1* is less than or equal to the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Less / Equal Unsigned (SLEU)	Compares two unsigned 32-bit values. If the contents of *src1* are less than or equal to the contents of *src2* (or *src1* is less than or equal to the immediate value) this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Less Than Unsigned (SLTU)	Compares two unsigned 32-bit values. If the contents of *src1* are less than the contents of *src2* (or *src1* is less than the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Not Equal (SNE)	Compares two 32-bit values. If the contents of *scr1* do not equal the contents of *src2* (or *src1* does not equal the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Shift Left Logical (SLL)	Shifts the contents of a register left (toward the sign bit) and inserts zeros at the least-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* or the immediate value specify the amount to shift. If *src2* (or the immediate value) is greater than 31 or less than 0, *src1* shifts by *src2* MOD 32.

Table 5-7 (continued) Computational Instruction Descriptions

Instruction Name	Description
Shift Right Arithmetic (SRA)	Shifts the contents of a register right (toward the least-significant bit) and inserts the sign bit at the most-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value) is greater than 31 or less than 0, *src1* shifts by the result of *src2* MOD 32.
Shift Right Logical (SRL)	Shifts the contents of a register right (toward the least-significant bit) and inserts zeros at the most-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value) is greater than 31 or less than 0, *src1* shifts by the result of *src2* MOD 32.
Subtract with Overflow (SUB)	Computes the twos-complement difference for two signed values. This instruction subtracts the contents of *src2* from the contents of *src1*, or it can subtract the contents of the immediate from the *src1* value. Subtract (with Overflow) puts the result in the destination register. When the true result's sign differs from the destination register's sign, the machine signals an overflow exception.
Subtract without Overflow (SUBU)	Computes the twos-complement difference for two 32-bit values. This instruction subtracts the contents of *src2* from the contents of *src1*, or it can subtract the contents of the immediate from the *src1* value. Subtract (without Overflow) puts the result in the destination register. Overflow exceptions never happen.
Trap if Equal (TEQ)	Compares two 32-bit values. If the contents of *src1* equal the contents of *src2* (or *src1* equals the immediate value), a trap exception occurs.
Trap if Not Equal (TNE)	Compares two 32-bit values. If the contents of *src1* do not equal the contents of *src2* (or *src1* does not equal the immediate value), a trap exception occurs.
Trap if Less Than (TLT)	Compares two signed 32-bit values. If the contents of *src1* are less than the contents of *src2* (or *src1* is less than the immediate value), a trap exception occurs.

Table 5-7 (continued) Computational Instruction Descriptions

Instruction Name	Description
Trap if Less Than Unsigned (TLTU)	Compares two unsigned 32-bit values. If the contents of *src1* are less than the contents of *src2* (or *src1* is less than the immediate value), a trap exception occurs.
Trap if Greater than or Equal (TGE)	Compares two signed 32-bit values. If the contents of *src1* are greater than the contents of *src2* (or *src1* is greater than the immediate value), a trap exception occurs.
Trap if Greater than or Equal Unsigned (TGEU)	Compares two unsigned 32-bit values. If the contents of *src1* are greater than the contents of *src2* (or *src1* is greater than the immediate value), a trap exception occurs.

Table 5-8 Computational Instruction Descriptions for MIPS3/4 Architecture

Instruction Name	Description
Doubleword Absolute Value (DABS)	Computes the absolute value of the contents of *src1*, treated as a 64-bit signed value, and puts the result in the destination register. If the value in *src1* is -2**63, the machine signals an overflow exception.
Doubleword Add with Overflow (DADD)	Computes the twos-complement sum of two 64-bit signed values. The instruction adds the contents of *src1* to the contents of *src2*, or it can add the contents of *src1* to the immediate value. When the result cannot be extended as a 64-bit number, the system signals an overflow exception.
Doubleword Add without Overflow (DADDU)	Computes the twos-complement sum of two 64-bit values. The instruction adds the contents of *src1* to the contents of *src2*, or it can add the contents of *src1* to the immediate value. Overflow exceptions never occur.
Doubleword Divide Signed (DDIV)	Computes the quotient of two 64-bit values. DDIV treats *src1* as the dividend. The divisor can be *src2* or the immediate value. It puts the quotient in the destination register. If the divisor is zero, the system signals an error and may issue a BREAK instruction. The DDIV instruction rounds towards zero. Overflow is signaled when dividing -2**63 by -1. **Note:** The special case DDIV *$0,src1,src2* generates the real doubleword divide instruction and leaves the result in the HI/LO register. The HI register contains the quotient. No checking for divide-by-zero is performed.

Table 5-8 (continued) Computational Instruction Descriptions for MIPS3/4 Architecture

Instruction Name	Description
Doubleword Divide Unsigned (DDIVU)	Computes the quotient of two unsigned 64-bit values. DDIVU treats *src1* as the dividend. The divisor can be *src2* or the immediate value. It puts the quotient in the destination register. If the divisor is zero, the system signals an exception and may issue a BREAK instruction. See note for DDIV concerning $0 as a destination. Overflow exceptions never occur.
Doubleword Multiply (DMUL)	Computes the product of two values. This instruction puts the 64-bit product of *src1* and *src2*, or the 64-bit product of *src1* and the immediate value, in the destination register.Overflow is not reported. **Note:** Use DMUL when you do not need overflow protection. It is often faster than DMULO and DMULOU. For multiplication by a constant, the DMUL instruction produces faster machine instruction sequences than DMULT or DMULTU can produce.
Doubleword Multiply (DMULT)	Computes the 128-bit product of two 64-bit signed values. This instruction multiplies the contents of *src1* by the contents of *src2* and puts the result in the HI and LO registers. No overflow is possible. **Note:** The DMULT instruction is a real machine language instruction.
Doubleword Multiply Unsigned (DMULTU)	Computes the product of two unsigned 64-bit values. It multiplies the contents of *src1* and the contents of *src2*, putting the result in the HI and LO registers. No overflow is possible. **Note:** The DMULTU instruction is a real machine language instruction.
Doubleword Multiply with Overflow (DMULO)	Computes the product of two 64-bit signed values. It puts the 64-bit product of *src1* and *src2*, or the 64-bit product of *src1* and the immediate value, in the destination register. When an overflow occurs, the system signals an overflow exception and may execute a BREAK instruction. **Note:** For multiplication by a constant, DMULO produces faster machine instruction sequences than DMULT or DMULTU can produce; however, if you do not need overflow detection, use the DMUL instruction. It is often faster than DMULO.

Table 5-8 (continued) Computational Instruction Descriptions for MIPS3/4 Architecture

Instruction Name	Description
Doubleword Multiply with Overflow Unsigned (DMULOU)	Computes the product of two 64-bit unsigned values. It puts the 64-bit product of *src1* and *src2*, or the 64-bit product of *src1* and the immediate value, into the destination register. When an overflow occurs, the system signals an overflow exception and may issue a BREAK instruction. **Note:** For multiplication by a constant, DMULOU produces faster machine instruction sequences than DMULT or DMULTU produces; however, if you do not need overflow detection, use the DMUL instruction. It is often faster than DMULOU.
Doubleword Negate with Overflow (DNEG)	Computes the negative of a 64-bit value. The instruction negates the contents of *src1* and puts the result in the destination register. If the value of *src1* is -2**63, the system signals an overflow exception.
Doubleword Negate without Overflow (DNEGU)	Negates the 64-bit contents of *src1* and puts the result in the destination register. Overflow is not reported.
Doubleword Remainder Signed (DREM)	Computes the remainder of the division of two signed 64-bit values. It treats *src1* as the dividend. The divisor can be *src2* or the immediate value. The DREMU instruction puts the remainder in the destination register. If the divisor is zero, the system signals an error and may issue a BREAK instruction.
Doubleword Remainder Unsigned (DREMU)	Computes the remainder of the division of two unsigned 64-bit values. It treats *src1* as the dividend. The divisor can be *src2* or the immediate value. The DREMU instruction puts the remainder in the destination register. If the divisor is zero, the system signals an error and may issue a BREAK instruction.
Doubleword Rotate Left (DROL)	Rotates the contents of a 64-bit register left (towards the sign bit). This instruction inserts in the least-significant bit any bits that were shifted out of the sign bit. The contents of *src1* specify the value to shift, and contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value) is greater than 63, *src1* shifts by *src2* MOD 64.

Table 5-8 (continued) Computational Instruction Descriptions for MIPS3/4 Architecture

Instruction Name	Description
Doubleword Rotate Right (DROR)	Rotates the contents of a 63-bit register right (towards the least-significant bit). This instruction inserts in the sign bit any bits that were shifted out of the least-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value is greater than 63, *src1* shifts by *src2* MOD 64.
Doubleword Shift Left Logical (DSLL)	Shifts the contents of a 64-bit register left (towards the sign bit) and inserts zeros at the least-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value) is greater than 63, *src1* shifts by *src2* MOD 64.
Doubleword Shift Right Arithmetic (DSRA)	Shifts the contents of a 64-bit register right (towards the least-significant bit) and inserts the sign bit at the most-significant bit. The contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value) is greater than 63, *src1* shifts by *src2* MOD 64.
Doubleword Shift Right Logical (DSRL)	Shifts the contents of a 64-bit register right (towards the least-significant bit) and inserts zeros at the most-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value) is greater than 63, *src1* shifts by *src2* MOD 64.
Doubleword Subtract with Overflow (DSUB)	Computes the twos-complement difference for two signed 64-bit values. This instruction subtracts the contents of *src2* from the contents of *src1*, or it can subtract the immediate value from the contents of *src1*. It puts the result in the destination register. When the true result's sign differs from the destination register's sign, the system signals an overflow exception.
Doubleword Subtract without Overflow (DSUBU)	Computes the twos complement difference for two unsigned 64-bit values. This instruction subtracts the contents of *src2* from the contents of *src1*, or it can subtract the immediate value from the contents of *src1*. It puts the result in the destination register. Overflow exceptions never happen.

Jump and Branch Instructions

The jump and branch instructions let you change an assembly program's control flow. This section of the book describes jump and branch instructions.

Jump and Branch Instructions

Jump and branch instructions change the flow of a program. Table 5-9 summarizes the formats of jump and branch instructions.

Table 5-9 Jump and Branch Format Summary

Description	Op-code	Operand
Jump	J	address
Jump and Link	JAL	*address* *target* *return,target*
Branch on Equal	BEQ	*src1,src2,label*
Branch on Greater	BGT	*src1,immediate,label*
Branch on Greater / Equal	BGE	
Branch on Greater / Equal Unsigned	BGEU	
Branch on Greater Than Unsigned	BGTU	
Branch on Less Than	BLT	
Branch on Less / Equal	BLE	
Branch on Less / Equal Unsigned	BLEU	
Branch on Less Than Unsigned	BLTU	
Branch on Not Equal	BNE	
Branch	B	label
Branch and Link	BAL	
Branch on Equal Likely*	BEQL	*src1,src2,label*

Table 5-9 (continued) Jump and Branch Format Summary

Description	Op-code	Operand
Branch on Greater Than Likely*	BGTL	*src1, immediate,label*
Branch on Greater / Equal Likely *	BGEL	
Branch on Greater / Equal Unsigned Likely*	BGEUL	
Branch on Greater Than Unsigned Likely*	BGTUL	
Branch on Less Than Likely*	BLTL	
Branch on Less / Equal Likely *	BLEL	
Branch on Less / Equal Unsigned Likely*	BLEUL	
Branch on Less Than Unsigned Likely*	BLTUL	
Branch on Not Equal Likely*	BNEL	
Branch on Equal to Zero	BEQZ	*src1,label*
Branch on Greater / Equal Zero	BGEZ	
Branch on Greater Than Zero	BGTZ	
Branch on Greater or Equal to Zero and Link	BGEZAL	
Branch on Less Than Zero and Link	BLTZAL	
Branch on Less / Equal Zero	BLEZ	
Branch on Less Than Zero	BLTZ	
Branch on Not Equal to Zero	BNEZ	
Branch on Equal to Zero Likely*	BEQZL	*src1,label*
Branch on Greater / Equal Zero Likely*	BGEZL	
Branch on Greater Than Zero Likely*	BGTZL	
Branch on Greater or Equal to Zero and Link Likely*	BGEZALL	
Branch on Less Than Zero and Link Likely*	BLTZALL	
Branch on Less / Equal Zero Likely*	BLEZL	

Table 5-9 (continued) Jump and Branch Format Summary

Description	Op-code	Operand
Branch on Less Than Zero Likely*	BLTZL	
Branch on Not Equal to Zero Likely*	BNEZL	

* Not valid inMIPS1 architecture.

Jump and Branch Instruction Descriptions

In Table 5-10 branch instructions, branch destinations must be defined in the source being assembled.

Table 5-10 Jump and Branch Instruction Descriptions

Instruction Name	Description
Branch (B)	Branches unconditionally to the specified label.
Branch and Link (BAL)	Branches unconditionally to the specified label and puts the return address in general register *$31*.
Branch on Equal (BEQ)	Branches to the specified label when the contents of *src1* equal the contents of *src2*, or when the contents of *src1* equal the immediate value.
Branch on Equal to Zero (BEQZ)	Branches to the specified label when the contents of *src1* equal zero.
Branch on Greater Than (BGT)	Branches to the specified label when the contents of *src1* are greater than the contents of *src2*, or it can branch when the contents of *src1* are greater than the immediate value. The comparison treats the comparands as signed 32-bit values.
Branch on Greater / Equal Unsigned (BGEU)	Branches to the specified label when the contents of *src1* are greater than or equal to the contents of *src2*, or it can branch when the contents of *src1* are greater than or equal to the immediate value. The comparison treats the comparands as unsigned 32-bit values.
Branch on Greater / Equal Zero (BGEZ)	Branches to the specified label when the contents of *src1* are greater than or equal to zero.

Table 5-10 (continued) Jump and Branch Instruction Descriptions

Instruction Name	Description
Branch on Greater / Equal Zero and Link (BGEZAL)	Branches to the specified label when the contents of *src1* are greater than or equal to zero and puts the return address in general register $31. When this write is done, it destroys the contents of the register. See the *MIPS Microprocessor User's Manual* appropriate to your architecture for more information. Do ***not*** use BGEZAL $31.
Branch on Greater or Equal (BGE)	Branches to the specified label when the contents of *src1* are greater than or equal to the contents of *src2*, or it can branch when the contents of *src1* are greater than or equal to the immediate value. The comparison treats the comparands as signed 32-bit values.
Branch on Greater Than Unsigned (BGTU)	Branches to the specified label when the contents of *src1* are greater than the contents of *src2*, or it can branch when the contents of *src1* are greater than the immediate value. The comparison treats the comparands as unsigned 32-bit values.
Branch on Greater Than Zero (BGTZ)	Branches to the specified label when the contents of *src1* are greater than zero.
Branch on Less Than Zero (BLTZ)	Branches to the specified label when the contents of *src1* are less than zero. The program must define the destination.
Branch on Less Than (BLT)	Branches to the specified label when the contents of *src1* are less than the contents of *src2*, or it can branch when the contents of *src1* are less than the immediate value. The comparison treats the comparands as signed 32-bit values.
Branch on Less / Equal Unsigned (BLEU)	Branches to the specified label when the contents of *src1* are less than or equal to the contents of *src2*, or it can branch when the contents of *src1* are less than or equal to the immediate value. The comparison treats the comparands as unsigned 32-bit values.
Branch on Less / Equal Zero (BLEZ)	Branches to the specified label when the contents of *src1* are less than or equal to zero. The program must define the destination.

Table 5-10 (continued) Jump and Branch Instruction Descriptions

Instruction Name	Description
Branch on Less or Equal (BLE)	Branches to the specified label when the contents of *src1* are less than or equal to the contents of *src2*, or it can branch when the contents of *src1* are less than or equal to the immediate value. The comparison treats the comparands as signed 32-bit values.
Branch on Less Than Unsigned (BLTU)	Branches to the specified label when the contents of *src1* are less than the contents of *src2*, or it can branch when the contents of *src1* are less than the immediate value. The comparison treats the comparands as unsigned 32-bit values.
Branch on Less Than Zero and Link (BLTZAL)	Branches to the specified label when the contents of *src1* are less than zero and puts the return address in general register $31. Because the value is always stored in register 31, there is a chance of a stored value being overwritten before it is used. See the MIPS microprocessor user's manual appropriate to your architecture for more information. Do ***not*** use BGEZAL $31
Branch on Not Equal (BNE)	Branches to the specified label when the contents of *src1* do not equal the contents of *src2*, or it can branch when the contents of *src1* do not equal the immediate value.
Branch on Not Equal to Zero (BNEZ)	Branches to the specified label when the contents of *src1* do not equal zero.
Jump (J)	Unconditionally jumps to a specified location. A symbolic address or a general register specifies the destination. The instruction J *$31* returns from a JAL call instruction.

Table 5-10 (continued) Jump and Branch Instruction Descriptions

Instruction Name	Description
Jump And Link (JAL)	Unconditionally jumps to a specified location and puts the return address in a general register. A symbolic address or a general register specifies the target location. By default, the return address is placed in register $31. If you specify a pair of registers, the first receives the return address and the second specifies the target. The instruction JAL *procname* transfers to *procname* and saves the return address. For the two-register form of the instruction, the target register may not be the same as the return-address register. For the one-register form, the target may not be $31.
Branch Likely Instructions	Same an the ordinary branch instruction (without the "Likely"), except in a branch likely instruction, the instruction in the delay slot is nullified if the conditional branch is not taken. **Note**: The branch likely instructions should be used only inside a *.set noreorder* schedule in an assembly program. The assembler does not attempt to schedule the delay slot of a branch likely instruction.

Special Instructions

The main processor’s special instructions do miscellaneous tasks. See Table 5-11.

Special Instruction Descriptions

Table 5-11 Special Instruction Descriptions

Instruction Name	Description
Break (BREAK)	Unconditionally transfers control to the exception handler. The *breakcode* operand is interpreted by software conventions. The *breakcode1* operand is used to fill the low-order 10 bits of the 20-bit immediate field in the BREAK instruction. The optional second operand, *breakcode2*, fills the high-order 10 bits.
Exception Return (ERET)	Returns from an interrupt, exception or error trap. Similar to a branch or jump instruction, ERET executes the next instruction before taking effect. Use this on R4000 processor machines in place of RFE.
Move From HI Register (MFHI)	Moves the contents of the HI register to a general-purpose register.
Move From LO Register (MFLO)	Moves the contents of the LO register to a general-purpose register.
Move To HI Register (MTHI)	Moves the contents of a general-purpose register to the HI register.
Move To LO Register (MTLO)	Moves the contents of a general-purpose register to the LO register.
Restore From Exception (RFE)	Restores the previous interrupt called and user / kernel state. This instruction can execute only in kernel state and is unavailable in user mode.
Syscall (SYSCALL)	Causes a system call trap. The operating system interprets the information set in registers to determine what system call to do.

Coprocessor Interface Instructions

The coprocessor interface instructions provide standard ways to access your machine's coprocessors. See Table 5-12 and Table 5-13.

Coprocessor Interface Summary

Table 5-12 Coprocessor Interface Formats

Description	Op-code	Operand
Load Word Coprocessor *z*	LWC*z*	*dest-copr,address*
Load Double Coprocessor *z**	LDC*z*	
Store Word Coprocessor *z*	SWC*z*	*src-copr, address*
Store Double Coprocessor *z**	SDC*z*	
Move From Coprocessor *z*	MFC*z*	*dest-gpr, source*
Move To Coprocessor *z*	MTC*z*	*src-gpr, destination*
Doubleword Move From Coprocessor *z* **	DMFC*z*	
Doubleword Move To Coprocessor *z* **	DMTC*z*	
Branch Coprocessor *z* False	BC*z*F	label
Branch Coprocessor *z* True	BC*z*T	
Branch Coprocessor *z* False Likely*	BC*z*FL	
Branch Coprocessor *z* True Likely*	BC*z*TL	
Coprocessor *z* Operation	C*z*	expression
Control From Coprocessor *z*	CFC*z*	*dest-gpr, source*
Control To Coprocessor z	CTC*z*	*src-gpr, destination*

* Not valid in MIPS1 architectures.

** Not valid in MIPS1 and MIPS2 architectures.

Note: You cannot use coprocessor load and store instructions with the system control coprocessor (cp0).

Coprocessor Interface Instruction Descriptions

Table 5-13 Coprocessor Interface Instruction Descriptions

Instruction Name	Description
Branch Coprocessor *z* True (BC*z*T)	Branches to the specified label when the specified coprocessor asserts a true condition. The *z* selects one of the coprocessors. A previous coprocessor operation sets the condition.
Branch Coprocessor *z* False (BC*z*F)	Branches to the specified label when the specified coprocessor asserts a false condition. The *z* selects one of the coprocessors. A previous coprocessor operation sets the condition.
Branch Coprocessor *z* True Likely (BC*z*TL)	Branches to the specified label when the specified coprocessor asserts a true condition. If the conditional branch is not taken, the instruction in the branch delay slot is nullified. **Note**: The branch likely instructions should be used only within a *.set noreorder* block. The assembler does not attempt to schedule the delay slot of a branch likely instruction.
Branch Coprocessor *z* False Likely (BC*z*FL)	Branches to the specified label when the specified coprocessor asserts a false condition. If the conditional branch is not taken, the instruction in the branch delay slot is nullified. **Note**: The branch likely instructions should be used only within a *.set noreorder* block. The assembler does not attempt to schedule the delay slot of a branch likely instruction.
Control From Coprocessor *z* (CFC*z*)	Stores the contents of the coprocessor control register specified by the source in the general register specified by **dest-gpr**.
Control To Coprocessor (CTC*z*)	Stores the contents of the general register specified by **src-gpr** in the coprocessor control register specified by the destination.
Coprocessor *z* Operation (*Cz*)	Executes a coprocessor-specific operation on the specified coprocessor. The *z* selects one of four distinct coprocessors.

Table 5-13 (continued) Coprocessor Interface Instruction Descriptions

Instruction Name	Description
Load Word Coprocessor *z* (LWC*z*)	Loads the destination with the contents of a word that is at the memory location specified by the effective address. The *z* selects one of four distinct coprocessors. Load Word Coprocessor replaces all register bytes with the contents of the loaded word. If bits 0 and 1 of the effective address are not zero, the machine signals an address exception.
Load Double Coprocessor *z* (LDC*z*)	Loads a doubleword from the memory location specified by the effective address and makes the data available to coprocessor unit *z*. The manner in which each coprocessor uses the data is defined by the individual coprocessor specifications. This instruction is not valid in MIPS1 architectures. If any of the three least-significant bits of the effective address are non-zero, the machine signals an address error exception.
Move From Coprocessor *z* (MFC*z*)	Stores the contents of the coprocessor register specified by the source in the general register specified by *dest-gpr*.
Move To Coprocessor *z* (MTC*z*)	Stores the contents of the general register specified by *src-gpr* in the coprocessor register specified by the *destination*.
Doubleword Move From Coprocessor *z* (DMFCz)	Stores the 64-bit contents of the coprocessor register specified by the source into the general register specified by *dest-gpr*.
Doubleword Move To Coprocessor *z* (DMTCz)	Stores the 64-bit contents of the general register *src-gpr* into the coprocessor register specified by the *destination*.

Table 5-13 (continued) Coprocessor Interface Instruction Descriptions

Instruction Name	Description
Store Word Coprocessor *z* (SWC*z*)	Stores the contents of the coprocessor register in the memory location specified by the effective address. The *z* selects one of four distinct coprocessors. If bits 0 and 1 of the effective address are not zero, the machine signals an address error exception.
Store Double Coprocessor *z* (SDC*z*)	Coprocessor z sources a doubleword, which the processor writes the memory location specified by the effective address. The data to be stored is defined by the individual coprocessor specifications. This instruction is not valid in MIPS1 architecture. If any of the three least-significant bits of the effective address are non-zero, the machine signals an address error exception.